

2

AD-A237 122



ION PAGE

Form Approved  
OMB No. 0704-0188

Pub  
ma  
inc  
VA

1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and information. Send comments regarding this burden estimate or any other aspect of this collection of information, services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, Virginia 22202-4302, and the Office of Management and Budget, Paperwork Project (0704-0188), Washington, DC 20503.

1. REPORT USE ONLY (Leave blank)

2. REPORT DATE

May 1991

3. REPORT TYPE AND DATES COVERED

Special Technical

4. TITLE AND SUBTITLE

Using Consistent Subcuts for Detecting Stable Properties

6. AUTHOR(S)

Keith Marzullo, Laura Sabel

5. FUNDING NUMBERS

NAG 2-593

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Keith Marzullo, Assistant Professor  
Department of Computer Science  
Cornell University

8. PERFORMING ORGANIZATION  
REPORT NUMBER

91-1205

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA/ISTO

10. SPONSORING/MONITORING  
AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE  
DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Please see page 1 of this technical report.

A-1

14. SUBJECT TERMS

15. NUMBER OF PAGES

12

16. PRICE CODE

17. SECURITY CLASSIFICATION  
OF REPORT

UNCLASSIFIED

18. SECURITY CLASSIFICATION  
OF THIS PAGE

UNCLASSIFIED

19. SECURITY CLASSIFICATION  
OF ABSTRACT


UNCLASSIFIED

20. LIMITATION OF  
ABSTRACT

UNLIMITED



91 6 17 050

91-02333  


**Using Consistent Subcuts  
for Detecting Stable Properties\***

Keith Marzullo  
Laura Sabel

TR 91-1205  
May 1991

Department of Computer Science  
Cornell University  
Ithaca, NY 14853-7501

---

\*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904, and by grants from IBM and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

# Using Consistent Subcuts for Detecting Stable Properties\*

Keith Marzullo

marzullo@cs.cornell.edu

Laura Sabel

sabel@cs.cornell.edu

Cornell University  
Department of Computer Science  
Ithaca, New York 14853  
May 17, 1991

## Abstract

We present a general algorithm for detecting whether a property holds in a distributed system, where the property is a member of a class we call the *locally stable properties*. Our algorithm is based on a decentralized method of constructing a maximal subset of the local states that are mutually consistent, which in turn is based on a weakened version of vector time stamps. We demonstrate the utility of our algorithm by using it to derive some specialized property-detection protocols, including two previously-known protocols that are known to be efficient.

[CL85] gives a simple algorithm that can be used to determine whether or not the global state of an asynchronous distributed system satisfies a given stable property. This algorithm is very general and can be used to detect *any* stable property of an asynchronous system. However, it is centralized and for most stable properties of interest, it is inefficient in the number of messages used.

In this paper, we present an algorithm that can be used to detect stable properties. This algorithm is general in that it can detect a wide class of stable properties (although not as wide as [CL85]), yet it is decentralized and can be optimized for different properties. We demonstrate its utility by using it to derive some specialized property-detection protocols, including two previously-known protocols that are known to be efficient.

## 1 Definitions

We consider an asynchronous distributed system consisting of a set of  $n$  nonfaulty processes  $P = \{p_1, p_2, \dots, p_n\}$ . Between any two processes  $p_i$  and  $p_j$  there exist two unidirectional fault-free FIFO

---

\*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904, and by grants from IBM and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

channels:  $C_{i,j}$  from  $p_i$  to  $p_j$  and  $C_{j,i}$  from  $p_j$  to  $p_i$ , and these channels have unbounded delivery time. Processes communicate only by sending and receiving messages over these channels.

A *global state*  $\Sigma \in \mathcal{G}$  is a consistent set of process states and channel states, defined more precisely in Equation 2, below. A *property* is a predicate expressed over the global state of the system. A *stable* property is an invariant: once it becomes true, it continues to be true. The most common examples of stable properties of distributed systems are deadlock of a subset of the processes, termination of a distributed computation, and the lack of a token among the processes. There are, of course, other stable properties of interest. For example, in a token passing system that can lose but not regenerate tokens, the predicate "there are no more than  $k$  tokens in the system" is a stable property.

Processes execute *events*, which can be send events, receive events, or local events. We say that an event is *relevant* to a property  $\Phi$  if the execution of the event can potentially affect  $\Phi$ . More precisely, if  $\Phi$  is a boolean formula on the global state of the system, and if event  $e_i$  of  $p_i$  changes a part of the state that is referenced in  $\Phi$ , then  $e_i$  is a relevant event. For example, if  $\Phi =$  "a subset of the processes are deadlocked" then the relevant events include those that request a resource and those that grant a resource, since both of these kinds of events affect whether the system is deadlocked. Note that these events could be local events, send events, or receive events, depending on exactly how  $\Phi$  is defined. Unless stated otherwise,  $e_i$  is an event of process  $p_i$ . Each event  $e_i$  results in the local state  $\sigma_i$  of  $p_i$ , and each local state  $\sigma_i$  has a corresponding event  $e_i$  that resulted in that state.

We will only be interested in detecting a subset of the stable properties, which we call the *locally stable* properties. Informally, a property  $\Phi$  is locally stable if no process involved in the property will change its state relative to  $\Phi$  once  $\Phi$  holds. For example, suppose  $\Phi =$  "processes  $p_i$  and  $p_j$  are deadlocked."  $\Phi$  is locally stable, because once  $\Phi$  becomes true, neither  $p_i$  nor  $p_j$  can execute an event that could affect  $\Phi$ ; in particular, requesting or granting a resource.

More formally, let  $\mathcal{G}$  be the set of all global states that the system can attain. Let  $\Sigma_\Phi$  be the portion of  $\Sigma$  that is referenced in  $\Phi$ , let  $A$  be a set of processes, and let  $\Sigma|A$  denote the subset of  $\Sigma$  that consists of the states of the processes in  $A$  and the channels between processes in  $A$ . Since  $\Phi$  is stable, if  $\Sigma$  satisfies  $\Phi$  then all states that are reachable from  $\Sigma$  also satisfy  $\Phi$ .<sup>1</sup> We will call  $\Phi$  *locally stable* if it satisfies the following condition: consider any  $\Sigma \in \mathcal{G}$  that satisfies  $\Phi$ , and let  $A$  be the set of processes that execute no relevant events in any state that is reachable from  $\Sigma$ . Then  $\Phi$  can be determined by only considering the values in  $\Sigma_\Phi|A$ . Note that  $A$  must be nonempty for properties that reference the state of the system; if  $A$  is empty, then  $\Phi$  can be determined without knowledge of the state of any process or channel and must therefore be constant. For this reason, we will assume in this paper that  $A$  is nonempty.

---

<sup>1</sup>  $\Sigma'$  is *reachable* from  $\Sigma$  if there is a valid execution that takes  $\Sigma$  to  $\Sigma'$ .

The most commonly-studied stable properties—deadlock, termination and no token—are all locally stable. For example, if  $\Sigma$  is a deadlock state, then  $A$  includes the deadlocked processes, and so the presence of deadlock can be determined by considering the states of the processes in  $A$ . The property “there are no more than  $k : k > 0$  tokens in the system” in a system where a token can be lost when passed is *not* a locally stable property. This is because if  $\Sigma$  is a state containing  $k$  tokens, then every process can execute a relevant event (namely, it can pass a token), and so  $A$  is empty. The condition cannot be detected from the values in  $\Sigma_\Phi|A$ , since there are no values in this set.

Our protocol will be based on a weak version of *vector clocks* [Mat89]. The usual definition of a vector clock  $V(e_i)$  is:

- $V(e_i)[i]$  is the number of events that  $p_i$  has executed through  $e_i$ , and
- $V(e_i)[j], j \neq i$  is the number of events that  $p_i$  knew that  $p_j$  had executed when  $p_i$  executed  $e_i$ .

This definition gives us the following two relations between vector clocks and cuts, where  $\rightarrow$  is the *happens-before* relation defined in [Lam78]. Equation 1 defines the happens-before relation in terms of vector clocks, and Equation 2 defines when a set of local states comprise a global state:

$$\forall i, j : i \neq j : V(e_i)[i] \leq V(e_j)[i] \equiv e_i \rightarrow e_j \quad (1)$$

$$\forall i, j : V(e_i)[i] \geq V(e_j)[i] \equiv \langle \sigma_1, \dots, \sigma_n \rangle \in \mathcal{G} \quad (2)$$

We weaken this definition to *weak vector clocks* in which the index  $V(e_i)[i]$  counts only the number of *relevant* events that  $p_i$  has executed through  $e_i$ . With weak vector clocks, several events of  $p_i$  may have the same value of  $V(e_i)[i]$ , but all such states result in the same local state with respect to  $\Phi$ . Let  $E(e_i)$  be the events of  $p_i$  that are equivalent to  $e_i$  in that they have the same weak vector clock  $V(e_i)$ . Similarly, let  $S(\Sigma)$  be the set of (not necessarily consistent) cuts that are equivalent to  $\Sigma$ ; that is, if  $\Sigma' = \{\sigma'_1, \dots, \sigma'_n\}$ , then  $\Sigma' \in S(\Sigma) \equiv \forall \sigma_i \in \Sigma : e'_i \in E(e_i)$ .

The following weakened versions of Equations 1 and 2 hold for both vector clocks and weak vector clocks:

$$\begin{aligned} \forall i, j : i \neq j : V(e_i)[i] \leq V(e_j)[i] \equiv \\ \exists e'_i, e'_j : e'_i \in E(e_i) \wedge e'_j \in E(e_j) \wedge e'_i \rightarrow e'_j \end{aligned} \quad (3)$$

$$\forall i, j : V(e_i)[i] \geq V(e_j)[i] \equiv \exists \Sigma \in \mathcal{G} : \Sigma \in S(\langle \sigma_1, \dots, \sigma_n \rangle) \quad (4)$$

The difference between vector clocks and weak vector clocks is illustrated in Figure 1. We assume that the predicate of interest references  $x$  and  $y$ , but not  $u$  nor any of the channel states. The upper execution shows normal vector clock values and the lower execution shows the weak

vector clock values. Note that although the events  $x := 1$  and  $y := 3$  do not form a consistent cut, their timestamps in (b) satisfy Equation 4 since there *does* exist a consistent cut in which  $(x = 1, y = 3)$ .

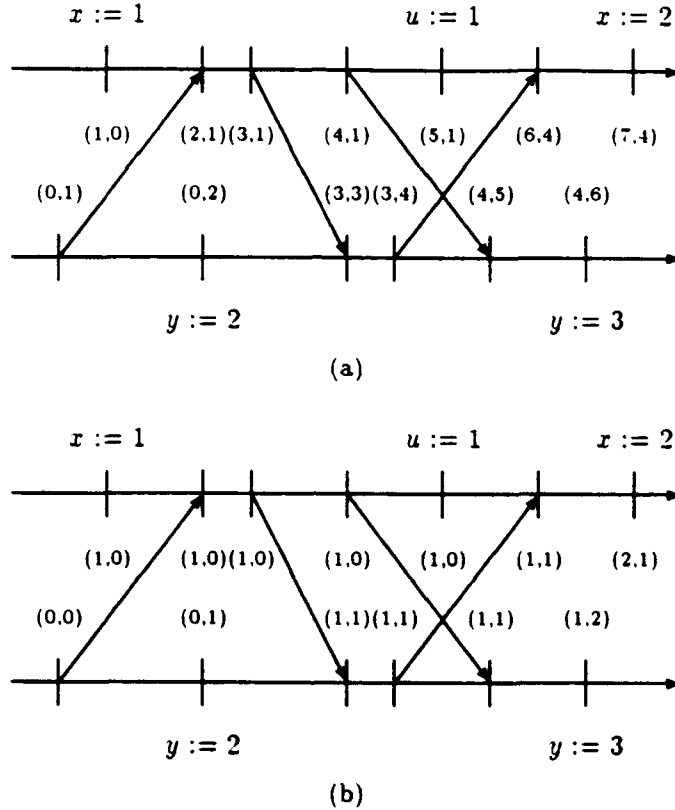


Figure 1: Execution: (a) with vector clocks, (b) with weak vector clocks.

## 2 Protocol

We first assume that a process  $p_0$  wishes to determine when the global state of the processes  $P = \{p_1, \dots, p_n\}$  satisfies a locally stable property  $\Phi$ . In Section 2.1, we will change this protocol so that any number of processes in  $P$  may concurrently assume the role of  $p_0$ . For simplicity, the state of the channel  $C_{i,j}$  from  $p_i$  to  $p_j$  will be represented by two (unbounded) queues:  $send_i[j]$ , which is the sequence of messages  $p_i$  has sent to  $p_j$  and is maintained by  $p_i$ , and  $recv_j[i]$ , which is the sequence of messages  $p_j$  has received from  $p_i$  and is maintained by  $p_j$ . Whenever we apply this algorithm, we will need to show that the length of these queues is in fact bounded by a small value.

Whenever a process  $p_i$  executes a relevant event  $e_i$ ,  $p_i$  records in a buffer  $B_i$  its state relative to

$\Phi$  and the vector time stamp  $V(e_i)$ . Thus, having executed  $e_i$ , the value of  $B_i$  will be  $(\sigma_i, V(e_i))$ . We will abbreviate these two components of  $B_i$  as  $B_i.\sigma$  and  $B_i.V$ . Then,  $p_0$  periodically collects the values of the buffers in any order. Once  $p_0$  has received these values,  $p_0$  determines if there exists a *maximal consistent subcut* among  $\{B_1, \dots, B_n\}$  that satisfies  $\Phi$ . By consistent subcut, we mean a set of states whose timestamps satisfy Equation 4; hence, the state of a single process is trivially a consistent subcut. If  $p_0$  can find such a subcut that satisfies  $\Phi$ , then  $\Phi$  must currently hold.<sup>2</sup>

Unfortunately, the number of maximal subcuts of a set of  $n$  weak vector clocks is  $\Omega(2^n)$ . Fortunately, it is not necessary for  $p_0$  to examine all of these subcuts. Suppose the set of buffer values contains  $B_i$  and  $B_j$  that are inconsistent:  $B_i.V[i] < B_j.V[i]$ . These two states violate Equation 4, and so cannot be part of the same consistent subcut. However,  $B_j$  records the fact that  $p_j$  has executed a relevant event since  $B_i$  was recorded. Since  $\Phi$  is locally stable, the event that generated  $B_i$  cannot have been involved in establishing  $\Phi$ , and so  $p_0$  need not consider any consistent subcut containing  $B_i$ . Given the partial order  $B_i \succ B_j \stackrel{\text{def}}{=} B_i.V[j] > B_j.V[j]$ ,  $p_0$  need only find the greatest elements of  $\succ$ , which can be done in  $\Omega(n^2)$  time. We call this subset the *latest subcut*. By Equation 4 the latest subcut is clearly a maximal subcut, and all events not in the latest subcut have executed relevant events since recording their state and so their values can be ignored.

The soundness of this protocol is straightforward. We now argue that the protocol is complete as well; that is, if  $\Phi$  holds, then our protocol will detect  $\Phi$ . Let  $\Sigma$  be the first global state in which  $\Phi$  holds. Since  $\Phi$  is locally stable, there is a nonempty set of processes  $A$  none of which execute a relevant event after  $\Sigma$ ; these processes will not change their states relative to  $\Phi$  or update their vector clocks after  $\Sigma$ . If  $p_0$  initiates the protocol after  $\Sigma$  (i.e., when  $\Phi$  holds), then  $p_0$  will collect the states  $\Sigma_\Phi|A$ . From the definition of  $\succ$ , the state of a process  $p_i$  in  $A$  must be in any latest subcut constructed by  $p_0$  because  $p_i$  will execute no relevant event. Hence,  $p_0$  will detect  $\Phi$ .

## 2.1 Optimization

In the above protocol,  $p_0$ 's role is to collect the states, determine the latest subcut and check if  $\Phi$  holds in this subcut. We can decentralize these steps by collecting the states in a token.

Consider a token  $K$  that consists of  $n$  entries  $\langle D_1, \dots, D_n \rangle$  where each entry  $D_i = (B_i.\sigma, B_i.V[i])$ ; that is,  $D_i$  will hold the state of  $p_i$  relevant to  $\Phi$  and the local component of  $p_i$ 's vector clock when it generated this state. Assume that there exists a special value  $\perp$  for  $D_i$  indicating that the state has not yet been collected; so, all of the  $D_i$  in  $K$  are initially set to  $\perp$ .

Whenever  $p_i$  wants to know whether  $\Phi$  holds, it generates an empty token  $K$ , inserts its state

<sup>2</sup>The initiator could examine all consistent subcuts, but if  $A' \subseteq A$  and  $\Sigma_\Phi|A'$  supports  $\Phi$ , then  $\Sigma_\Phi|A$  will also support  $\Phi$ , so we need only examine the maximal subcuts. Of course,  $\Phi$  may be of the form  $\forall p_i : \Psi(p_i)$ , in which case only a full consistent cut will satisfy  $\Phi$ .

into  $D_i$ , and passes the token to any other process. When a process  $p_j$  receives a token  $K$ , it takes the following steps:

- $p_j$  sets  $D_j$  to  $(B_j.\sigma, B_j.V[j])$ .
- $p_j$  casts out any values  $D_i$  that are not part of the latest subcut. Note that by definition,  $B_j$  must be part of the latest subcut, so only the earlier values  $D_i$  need be tested with respect to  $B_j$ . From above, the value  $B_i$  can be discarded if  $B_i.V[i] < B_j.V[i]$ . The value  $B_i.V[i]$  is stored in  $D_i$ , so  $K$  carries enough information for  $p_j$  to make this test. If  $D_i$  is not in the latest subcut, then  $p_j$  sets  $D_i$  to  $\perp$ .
- $p_j$  determines whether the values  $D_i$  satisfy  $\Phi$ . If so, then the detection is made; otherwise,  $p_j$  forwards the token to a process  $p_k$ , chosen fairly, with  $D_k = \perp$ . If there is no such process, then  $p_j$  can drop the token.

Note that we have no *a priori* restriction on how many tokens there can be in the system at any time or on how the token is passed, other than it is passed in a fair manner. These decisions can be made when the algorithm is applied to a particular problem.

### 3 Termination Detection

We can now instantiate the general protocol given above to obtain a protocol that detects *termination* in a distributed system. There are many variations of this property. The earliest that we know of is due to Dijkstra, in [Dij80]. The following definition is the same as that given in [Mis83].

All processes are either *active* or *idle*. Only active processes can send messages. An active process may become idle at any time; an idle process may become active upon receipt of a message. The system is *terminated* when all processes in the system are idle and there are no messages in transit.

The events that are relevant to termination are sending a message, receiving a message, becoming idle, and becoming active. Therefore, each process will update its (weak) vector clock upon executing any of these events. The state of a process relative to termination consists of whether the process is active or idle and whether there is a message on an incoming channel. Note that for this problem, we do not need to keep track of the contents of the messages exchanged between processes; only the number of messages is important. To capture the channel states, we have each process keep track of how many messages it has sent and received on each adjacent channel. The combined information of all of the processes will then yield the number of messages in transit on each channel: if  $p_i$  has sent more messages to  $p_j$  than  $p_j$  has received from  $p_i$ , then there is at least one message on channel  $C_{i,j}$ .

We instantiate the general protocol given in Section 2.1 as follows:

Each process  $p_i$  maintains the following local state variables:

- $active_i$ : Boolean = true if and only if  $p_i$  is active.
- $send_i[1..n]$ : Integer array.  $send_i[j]$  = the number of messages that  $p_i$  has sent to  $p_j$  (initially 0).
- $recv_i[1..n]$ : Integer array.  $recv_i[j]$  = the number of messages that  $p_i$  has received from  $p_j$  (initially 0).

When  $p_i$  sends a message to  $p_j$ ,  $send_i[j]$  is incremented. When  $p_i$  receives a message from  $p_j$ ,  $recv_i[j]$  is incremented. When  $p_i$  becomes active or idle,  $active_i$  is set appropriately.

At some point, an idle process  $p_u$  will start the detection algorithm by circulating a token as described in Section 2.1. The termination condition can only be evaluated over a total global state, so a positive determination can only be made by the process  $p_f$  that is the last to add its state to the token.

Process  $p_f$  detects termination if and only if the following three conditions hold:

1. The timestamps in the token form a consistent cut over all processes;
2. All processes are idle:  $\forall i : active_i = \text{false}$ ;
3. There are no messages in transit:  $\forall i, j : send_i[j] = recv_j[i]$ .

We claim that item 1 is redundant: item 3 implies that the cut is consistent. Suppose by way of contradiction that when all of the states and timestamps have been collected, item 3 holds but the timestamps form an inconsistent cut. That the cut is inconsistent implies from Equation 4 that for some  $i, j$ ,  $B_i.V[i] < B_j.V[i]$ .  $B_j.V[i]$  is advanced only when  $p_j$  receives a message and events local to  $p_j$  only affect  $B_j.V[j]$ . Therefore, there must have been a chain of messages between  $p_i$  and  $p_j$  between the time that  $B_i$  was collected and the time that  $B_j$  was collected. This implies that there is some  $k$  such that  $send_i[k] < recv_k[i]$ . This contradicts the assumption that item 3 holds.

Therefore,  $p_f$  need only check the last two items. In fact, these checks can be done incrementally. For example, we can assign a total order to the processes and have the token passed along that total order. When process  $p_k$  receives the token, it tests to see if

$$\neg active_k \wedge (\forall \ell : 1 \leq \ell < k : (send_k[\ell] = recv_\ell[k]) \wedge (send_\ell[k] = recv_k[\ell])).$$

If this condition does not hold, then  $p_k$  can drop the token. If the condition holds and  $k = n$ , then termination is detected; otherwise,  $p_k$  fills in  $D_k$  and passes the token to  $p_{k+1}$ .

This yields the protocol given in [Mat87] as the *channel counting* protocol, which only requires  $n$  messages to detect termination once it holds, and which can be further refined into a protocol that is space-efficient.

## 4 Deadlock Detection

We now instantiate the general protocol given in Section 2 to obtain a protocol that detects *k-out-of-m deadlock* in a distributed system. This problem was first formulated and solved in [BTS4]. In this formulation, a process can request  $k$  resources from a pool of  $m$  resources.

A process is either active or blocked. An active process is one that is not waiting for any other process. Active processes may issue *k-out-of-m* requests in the following way. When an active process  $p_i$  requires  $k$  processes to carry out some request, it sends **request** messages to each of the  $m$  processes that can perform this action. Process  $p_i$  then becomes blocked, and waits until the action requested is carried out by at least  $k$  of the  $m$  processes. A process can not send any further requests while blocked.

Only active processes can carry out a requested action. If a process  $p_j$  receives a request while active, it will either become blocked or carry out  $p_i$ 's requested action within finite time. In the latter case,  $p_j$  will send a **grant** message to  $p_i$ . When  $p_i$  receives  $k$  **grant** messages, it becomes active again. It then relinquishes the requests made to the rest of the processes to which it sent **request** messages by sending them **relinquish** messages. We assume that a **grant** message identifies its corresponding **request** message (for example, by using a sequence number) so that if  $p_i$  receives more than  $k$  **grant** messages for a given **request** message, the extra **grant** messages can be discarded.

The global state of the system will be represented as follows. Each process  $p_i$  maintains the following variables:

- $k_i$ : Integer = the number of **grant** messages required for  $p_i$  to become active (initially 0).
- $r\_send_i[1..n]$ : Integer array.  $r\_send_i[j]$  is the number of **request** messages that  $p_i$  has sent to  $p_j$  (initially 0).
- $r\_recv_i[1..n]$ : Integer array.  $r\_recv_i[j]$  is the number of **request** messages that  $p_i$  has received from  $p_j$  (initially 0).
- $g\_send_i[1..n]$ : Integer array.  $g\_send_i[j]$  is the number of **grant** messages that  $p_i$  has sent to  $p_j$  (initially 0).
- $g\_recv_i[1..n]$ : Integer array.  $g\_recv_i[j]$  is the number of **grant** messages that  $p_i$  has received from  $p_j$  (initially 0).

We also define the following two state functions:

- $blk_i$ : Integer set.  $j \in blk_i$  if  $p_i$  has received a **request** message from  $p_j$  and  $p_i$  has not sent a corresponding **grant** message (i.e.,  $p_i$  is blocking  $p_j$ ). This is defined as

$$j \in blk_i \stackrel{\text{def}}{=} r\_recv_i[j] > g\_send_i[j]$$

- $wf_i$ : Integer set.  $j \in wf_i$  if  $p_i$  has sent a **request** message to  $p_j$ , and  $p_i$  has not received a corresponding **grant** message (i.e.,  $p_i$  is waiting for  $p_j$ ). This is defined as

$$j \in wf_i \stackrel{\text{def}}{=} r\_send_i[j] > g\_recv_i[j]$$

The system wait-for graph is constructed as follows:

- a *waits-for* edge is drawn from  $p_i$  to  $p_j$  when  $j \in wf_i \wedge (i \in blk_j \vee (r\_send_i[j] > r\_recv_j[i]))$ ;
- the value  $\kappa_i$  is defined as  $k_i - |\forall j : g\_send_j[i] - g\_recv_i[j]|$ .

Deadlock is tested by reducing this graph: if an edge points from  $p_i$  to  $p_j$  and  $p_j$  is active, then the edge can be erased and  $\kappa_i$  can be reduced by one; and if a process has  $\kappa_i = 0$ , then all of its outgoing edges can be erased. The system is deadlocked if and only if there are edges that cannot be removed by following these two rules.

The relevant events are requesting a resource, granting a resource, receiving a grant, and receiving a request. Several actions may be associated with a relevant event; for example, when  $p_i$  requests 1 out of 2 resources from  $p_j$  and  $p_k$ , the following steps are executed atomically:

1.  $k_i$  is set to 1;
2.  $r\_send_i[j]$  and  $r\_send_i[k]$  are incremented;
3.  $B_i.V[i]$  is incremented.

The **request** messages can then be sent to  $p_j$  and  $p_k$ .

As described in Section 2.1, a process can start circulating a token at any time. For deadlock, a process need only send a token if it is blocked for an excessive time, and a logical place to forward the token is to one of the processes upon which it is blocked.

This protocol can be optimized further. For example, if we restrict ourselves to RPC deadlock (1-out-of-1 requests), then  $k_i = 1$  and need not be represented in the wait-for graph, and the wait-for graph is reducible if and only if it does not contain a cycle. Hence, when a process  $p_i$  receives a token  $K$  it can test for a cycle in the wait-for graph simply by testing to see if its state is still consistent with  $D_i$ . Furthermore, if a blocked process delays receiving any **request** messages while blocked, then it is easy to show that the vector clocks are not necessary: all states in the token are consistent at any time. Recall that when  $p_j$  receives a token from  $p_i$  where  $i \in blk_j$ ,  $p_j$  adds its state and forwards the token to the process in  $wf_j$  if  $wf_j$  is nonempty and drops the token if  $wf_j$  is empty. Suppose by way of contradiction that  $D_i$  and  $D_j$  are two entries in the token such that  $B_i$  and  $B_j$  are inconsistent:  $B_i.V[i] < B_j.V[i]$ . Then  $p_i$  must have sent a **request** message since its state was added to the token. Furthermore,  $B_j$  must have been added to the token after  $B_i$ , which

implies that there is a path in the wait-for graph from  $p_i$  to  $p_j$ . But this means that  $p_i$  cannot become active until  $p_j$  sends a **grant** message to the process in  $blk_j$ , contradicting that  $p_i$  sent a **request** message since its state was added to the token. Therefore, all states in the token at any time are consistent.

A similar argument can be made to show that this protocol will detect **and**-deadlock ( $m$ -out-of- $m$  requests), but the argument is more complex. The resulting protocol is the one presented in [CMH83].

## 5 Conclusion

This paper presents a general protocol for detecting a class of stable properties (the *locally stable* properties) by constructing consistent subcuts. The protocol collects the consistent subcuts in a decentralized manner and is message efficient. We have demonstrated its use by refining it to a known protocol for termination detection, a new protocol for  $k$ -out-of- $m$  deadlock detection, and a known protocol for **and**-deadlock detection. It is interesting to note that the two known protocols are, in fact, implicitly constructing consistent subcuts.

The class of locally stable properties was defined in proving the protocol correct. We are interested in whether the protocol can be extended to detect a wider set of stable properties. We would also like to better understand the notion of relevant events and weak vector clocks. We have attempted to refine our protocol to several known protocols, and have found that subtle changes in the definition of relevant events and propagation of vector time stamps can greatly ease the process of refinement.

**Acknowledgements** We would like to thank Özalp Babaoğlu, Gil Neiger, Fred Schneider, and Sam Toueg for their contributions to the ideas in this paper. We would also like to thank Navin Budhiraja, Tushar Chandra, and Mark Wood for their valuable comments on earlier drafts of this paper.

## References

- [BT84] G. Bracha and Sam Toueg. A distributed algorithm for generalized deadlock detection. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 285–301. ACM SIGPLAN/SIGOPS, August 1984.
- [CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

- [CMH83] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.
- [Dij80] Edsger W. Dijkstra. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [Mat87] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.
- [Mat89] Friedemann Mattern. Time and global states of distributed systems. In Michel Cosnard et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, October 1989.
- [Mis83] Jayadev Misra. Detecting termination of distributed computations using markers. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 290–294. ACM SIGPLAN/SIGOPS, August 1983.